CS1010S Programming Methodology

Lecture 11

Memoization, Dynamic Programming & Exception Handling

8 April 2015

Today's Agenda

- Memoization
- Dynamic Programming
- Exception Handling

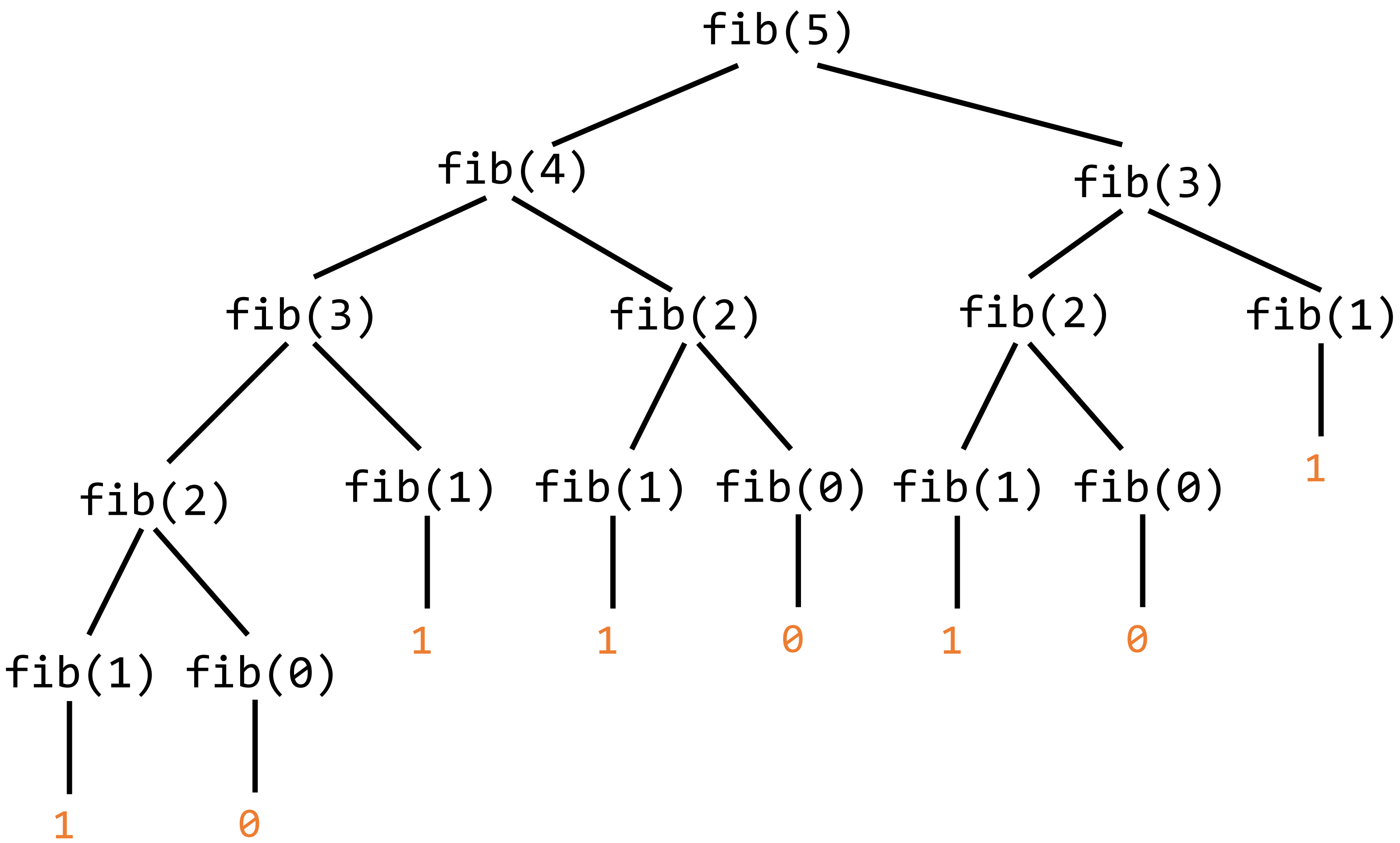
Recall: Fibonacci

```
def fib(n):
    if  n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Time complexity = $O(\Phi^n)$ (exponential!)

How can we do better?

Computing Fibonacci



Computing Fibonacci

fib 5

fib 4 fib 3

fib 3 fib 2 fib 2 fib 1

fib 2 fib 1 fib 1 fib 0 fib 1 fib 0 1

fib 1 fib 0 1 1 0 1 0

1 0

Lots of duplicates!

What's the obvious way to do better?

Remember what you had earlier computed!

Memoization

Notice the spelling,

NOT memorization

Simple Idea!!

Function records, in a table, values that have previously been computed.

- A memoized function:
  - Maintains a table in which values of previous calls are stored
  - Use the arguments that produced the values as keys

- When the memoized function is called, check table to see if the value exists:
  - If so, return value.
  - Otherwise, compute new value in the ordinary way and store this in the table.

Implementing Memoization

```
memoize_table = {}
def memoize(f, name):
    if name not in memoize_table:
        memoize_table[name] = {}
    table = memoize_table[name]
    def helper(*args):
        if args in table:
            return table[args]
        else:
            result = f(*args)
            table[args] = result
            return result
    return helper
```

Name to store in reference table

Fibonacci with Memoization

- Now that we have memoize, the obvious thing to do is:

```
memo_fib = memoize(fib, "fib")
```

- What's the time complexity now?

Still exponential!

HUH??

Fibonacci with Memoization

- Now that we have memoize, the obvious thing to do is:

```
memo_fib = memoize(fib,"fib")
```

- Problem: recursive step in fib will call `fib` instead of `memo_fib`.

Doing it Right!

```
def memo_fib(n):
    def helper(n):
        if  n==0:
            return 0
        elif n==1:
            return 1
        else:
            return memo_fib(n-1) + 
memo_fib(n-2)
    return memoize(helper, "memo_fib")(n)
```

Why??

What’s the time complexity now? $O(n)$ (linear)!

Doing it Right!

```
def memo_fib(n):
    def helper(n):
        if n==0:
            return 0
        elif n==1:
            return 1
        else:
            return memo_fib(n-1) +
                   memo_fib(n-2)
    return memoize(helper, "memo_fib")(n)
```

Each `fib(n)` is computed only once!

Doing it Right!

```
def memo_fib(n):
    def helper(n):
        if  n==0:
            return 0
        elif n==1:
            return 1
        else:
            return memo_fib(n-1) + memo_fib(n-2)
    return memoize(helper, "memo_fib")(n)
```

Efficiency of table lookup is important: Table lookup should be O(1), i.e. hash table.

What happens to time complexity if table lookup is not constant, say O(n)?

Another Example: C_k^n

```
def choose(n, k):
    if k > n:
        return 0
    elif k==0 or k==n:
        return 1
    else:
        return choose(n-1,k) +
                choose(n-1,k-1)
```

Why is the recursion true?

Remember Count-Change?

- Consider one of the elements x. x is either chosen or it is not.
- Then number of ways is sum of:
  - Not chosen. Ways to choose k elements out of remaining n−1 elements; and
  - Chosen. Ways to choose k −1 elements out of remaining n−1 elements

Another Example: C_k^n

```
def choose(n,k):
    if k > n:
        return 0
    elif k==0 or k==n:
        return 1
    else:
        return choose(n-1,k) +
               choose(n-1,k-1)
```

What is the order of growth?

How can we speed up the computation?

Memoization!

Memoized Choose

```
def memo_choose(n,k):
    def helper(n,k):
        if k > n:
            return 0
        elif k==0 or k==n:
            return 1
        else:
            return memo_choose(n-1,k) +
                   memo_choose(n-1,k-1)
    return memoize(helper,
                   "choose")(n,k)
```

Don’t need to use memoize function. Can just use a dictionary!

Chocolate Packing

- Suppose we are at a chocolate candy store and want to assemble a kilogram box of chocolates.
- Some of the chocolates (such as the caramels) at this store are absolutely the best, and others are only so-so.

Chocolate Packing

- We rate each one on a scale of 1 to 10, with 10 being the highest.
- Each piece of chocolate has a weight; for example, a caramel-flavoured chocolate weighs 13 grams.
- How do we put together the best box weighing at most 1 kilogram?

Abstract Data Type for Chocolates

```
def make_chocolate(desc,weight,value):
    return (desc,weight,value)

def get_description(choc):
    return choc[0]
def get_weight(choc):
    return choc[1]
def get_value(choc):
    return choc[2]
```

Here are the Chocolates

```
shirks_chocolates =
  (make_chocolate('caramel dark', 13, 10),
   make_chocolate('caramel milk', 13, 3),
   make_chocolate('cherry dark', 21, 3),
   make_chocolate('cherry milk', 21, 1),
   make_chocolate('mint dark', 7, 3),
   make_chocolate('mint milk', 7, 2),
   make_chocolate('cashew-cluster dark', 8, 6),
   make_chocolate('cashew-cluster milk', 8, 4),
   make_chocolate('maple-cream dark', 14, 1),
   make_chocolate('maple-cream milk', 14, 1))
```

Implementing a Box

```
def make_box(list_of_choc,weight,value):
    return (list_of_choc,weight,value)

def make_empty_box():
    return make_box((),0,0)

def box_chocolates(box):
    return box[0]
def box_weight(box):
    return box[1]
def box_value(box):
    return box[2]
```

Implementing a Box

```
def add_to_box(choc,box):
    return (box_chocolates(box)+(choc,),
            box_weight(box)
            + get_weight(choc),

box_value(box)+get_value(choc))

def better_box(box1,box2):
    if box_value(box1) > box_value(box2):
        return box1
    else:
        return box2
```

How to Solve this Problem?

- Enumerate all the possible boxes (constrained by weight limit)
- Compute desirability of each packing
- Pick box with highest desirability

Simple Solution

```
def pick(chocs,weight_limit):
    if chocs == () or weight_limit==0:
        return make_empty_box()
    elif get_weight(chocs[0]) > weight_limit: # 1st too
heavy
        return pick(chocs[1:],weight_limit)
 else:
        # none of 1st kind
        box1 = pick(chocs[1:],weight_limit)

      # at least one of 1st kind
        box2 = add_to_box(chocs[0],
                          pick(chocs,
                               weight_limit
                               - get_weight(chocs[0])))
        return better_box(box1,box2)
```

Simple Solution

- What is the order of growth?
  - Exponential! $O(2^n)$

- Again, a lot of repeat computations.

- Think memoization!

Memoized Version

```
def memo_pick(chocs,weight_limit):
    def helper(chocs,weight_limit):
        if chocs == () or weight_limit==0:
            return make_empty_box()
        elif get_weight(chocs[0]) > weight_limit:
            return memo_pick(chocs[1:],weight_limit)
        else:
            box1 = memo_pick(chocs[1:],weight_limit)
            box2 = add_to_box(chocs[0],
                     memo_pick(chocs,
                               weight_limit-
get_weight(chocs[0])))
            return better_box(box1,box2)
    return memoize(helper,"pick")(chocs,weight_limit)
```

Recap: Memoization

- Two Steps:
  - 1 Write function to perform required computation
  - 2 Add wrapper that stores the result of the computation
  - (lookup done with $O(1)$ lookup table)

- When you wrap your function, just make sure that he recursive calls go to the wrapped version and not the raw form.

Homework

Re-factor the chocolate packing code into OOP format.

Design Pattern: Wrapper (also called Decorator)

- Memoization as an idea is simply to remember the stuff you have done before so that you don’t do the same thing twice
- However, the method that we used to implement memoization is also an important concept

Design Pattern: Wrapper (also called Decorator)

- Design Pattern: Wrapper (also known as Decorator)
- Key idea is that you add an extra layer to introduce additional functionality and use the original function to do “old work”

Revisting memo_choose

- Consider `memo_choose(9, 4)`
- The following is the table of values at the end of the computation:

#f	#f	#f	#f	#f
1	1	#f	#f	#f
1	2	1	#f	#f
1	3	3	1	#f
1	4	6	4	1
1	5	10	10	5
#f	6	15	20	15
#f	#f	21	35	35
#f	#f	#f	56	70

Recall Pascal's Triangle

- If we were to fill up the table, we expect

1	0	0	0	0
1	1	0	0	0
1	2	1	0	0
1	3	3	1	0
1	4	6	4	1
1	5	10	10	5
1	6	15	20	15
1	7	21	35	35
1	8	28	56	70

Recall Pascal's Triangle

- If we were to fill up the table, we expect

1	0	0	0	0
1	1	0	0	0
1	2	1	0	0
1	3	3	1	0
1	4	6	4	1
1	5	10	10	5
1	6	15	20	15
1	7	21	35	35
1	8	28	56	70

Dynamic Programming

- Idea: why don't we compute choose by filling up this table from the bottom?
- Fancy name for this simple idea — Dynamic Programming :-)
- What is the order of growth then?

$$O(n)$$

Why??

Dynamic Programming: choose

```
def dp_choose(n,k):
    row = [1]*(k+1)
    table = []
    for i in range(n+1):
        table.append(row.copy())

    for j in range(1,k+1):
        table[0][j] = 0

    for i in range(1,n+1):
        for j in range(1,k+1):
            table[i][j] = table[i-1][j-1]
                        + table[i-1][j]
    return table[n][k]
```

allocate space for table (all 1's)

Fill first row with 0’s

fill the rest

return answer

Let’s check out the table

```
row = [1] * (k+1)

table = []
for i in range(n+1):
  table.append(
            row.copy())
```

$\xrightarrow{k+1}$

1 1 1 1 1

Let’s check out the table

```
row = [1] * (k+1)

table = []
for i in range(n+1):
  table.append(
                row.copy())
```

C_0^0 k+1 C_k^0

n+1

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

C_0^n C_k^n

Let’s check out the table

```
for j in range(1,k+1):
  table[0][j] = 0
```

	j			
1	0	0	0	0
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Let's check out the table

```
for j in range(1,k+1):
  table[0][j] = 0
```

1	0	0	0	0
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Let’s check out the table

1	0	0	0	0
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Let’s check out the table

```
for i in range(1,n+1):
  for j in
    range(1,k+1):

table[i][j] =
 table[i-1][j-1]
  + table[i-1][j]
```

$$
\begin{array}{c|ccccc}
 & & j & & & \\
\hline
 & 1 & 0 & 0 & 0 & 0 \\
 & 1 & 1 & 1 & 1 & 1 \\
i & 1 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 & 1
\end{array}
$$

Let’s check out the table

```
for i in range(1,n+1):
  for j in
    range(1,k+1):

table[i][j] =
 table[i-1][j-1]
  + table[i-1][j]
```

j

i

1	0	0	0	0
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Let’s check out the table

```
for i in range(1,n+1):
  for j in
    range(1,k+1):

table[i][j] =
 table[i-1][j-1]
  + table[i-1][j]
```

$$
\begin{array}{c|ccccc}
 & j \rightarrow & & & & \\
\hline
i \downarrow & 1 & 0 & 0 & 0 & 0 \\
 & 1 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 & 1
\end{array}
$$

Let’s check out the table

```
for i in range(1,n+1):
  for j in
    range(1,k+1):

table[i][j] =
 table[i-1][j-1]
  + table[i-1][j]
```

j

i

1	0	0	0	0
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Let’s check out the table

```
for i in range(1,n+1):
  for j in
    range(1,k+1):

table[i][j] =
 table[i-1][j-1]
  + table[i-1][j]
```

j

i

1	0	0	0	0
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Let’s check out the table

```
for i in range(1,n+1):
  for j in
    range(1,k+1):

table[i][j] =
 table[i-1][j-1]
  + table[i-1][j]
```

j →

i ↓

1	0	0	0	0
1	1	1	1	1
1	2	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Fast Forward

1	0	0	0	0
1	1	0	0	0
1	2	1	0	0
1	3	3	1	0
1	4	6	4	1
1	5	10	10	5
1	6	15	20	15
1	7	21	35	35

Can we adopt a dynamic programming approach for solving the chocolate packing problem?

YES (of course)!

How?

Let’s take another look at the chocolate packing problem…

Remember the table .... dynamic programming is about filling up some table efficiently so that the total time complexity is O(size of table)

What does the chocolate packing table look like?

- Table of list of chocolates vs weight limit
- Each table entry is the optimal choice for a given list and weight limit.

What does the chocolate packing table look like?

- Observation: If we have no chocolates, answer is, easy: Empty set ().
- Otherwise, the answer for a list of x types of chocolates is the better between:
  1. Additional chocolate + Optimal choice with remaining x types of chocolates and reduced weight limit
  2. Optimal choice with remaining x −1 types of chocolates and current weight limit.

Building the chocolate packing table

Idea: build a table from the smaller cases to larger cases

Weight Limit	()			
0	()			
1	()			
2	()			
3	()			
4	()			
5	()			
6	()			
7	()			
8	()			
…	()			

Building the chocolate packing table

Weight Limit	()	((A 4 2))		
0	()	()		
1	()	()		
2	()	() +A		
3	()	()		
4	()	(A)		
5	()	(A)		
6	()	(A) +A		
7	()	(A)		
8	()	(A,A)		
⋮	()			

⋮

Building the chocolate packing table

Weight Limit	()	((A 4 2))	((A 4 2) (B 5 3))	
0	()	()	()	
1	()	()	()	
2	()	()	() Cannot	
3	()	()	() add B	
4	()	(A)	(A)	
5	()	(A)	(B)	
6	()	(A)		
7	()	(A)		
8	()	(A,A)		
⋮	()	⋮		

Building the chocolate packing table

Weight Limit	()	((A 4 2))	((A 4 2) (B 5 3))	
0	()	()	()	
1	()	()	()	
2	()	()	()	
3	()	()	()	
4	()	(A)	(A) +B	
5	()	(A)	B	
6	()	(A)	B +B	
7	()	(A)	B	
8	()	(A,A)		
⋮	()	⋮		

Building the chocolate packing table

Weight Limit	()	((A 4 2))	((A 4 2) (B 5 3))	
0	()	()	()	
1	()	()	()	
2	()	()	()	
3	()	()	()	
4	()	(A)	(A)	
5	()	(A)	B	
6	()	(A)	B	
7	()	(A)	B +B	
8	()	(A,A)	(A,A)	
⋮	()	⋮	⋮	

Question of the Day: Write `dp_pick_chocolates` over the weekend. ☺

Prime Numbers

- In recitation, we defined a function `is_prime` to check whether a number is prime
- But what if we wanted to list ALL of the numbers that are prime, in the interval $[0, \ldots, n]$?

Prime Numbers: Naïve Soln

```
def is_prime(n): # O(n^0.5)
    if n == 0 or n == 1:
        return False
    elif n == 2:
        return True
    for i in range(2, int(sqrt(n))+1):
        if n % i == 0:
            return False
    return True

def naive_prime(n): # O(n^1.5)
    return [is_prime(i) for i in range(n+1)]
```

Prime Numbers

Idea: why don’t we compute the primes by filling up a table from the bottom?

Prime Numbers: DP

```
def dp_prime(n):
    bitmap = [True for i in range(n+1)]
    bitmap[0] = False # 0 is not prime
    bitmap[1] = False # 1 is not prime
    for i in range(2,n):
        if bitmap[i] == True:
            for j in range(2*i,n+1,i):
                bitmap[j] = False
    return bitmap
```

Prime Numbers: DP

How does it work?

~~0~~, 1, 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13

0, 1, 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13

0, 1, 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13

0, 1, 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13

0, 1, 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13

Errors and Exceptions

- Until now error messages haven’t been more than mentioned, but you have probably seen some
- Two kinds of errors (in Python):
  1. syntax errors
  2. exceptions

Syntax Errors

```
>>> while True print('Hello world')
SyntaxError: invalid syntax
```

Exceptions

- Errors detected during execution are called exceptions
- Examples:
  - ZeroDivisonError,
  - NameError,
  - TypeError

ZeroDivisionError

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in
<module>
    10 * (1/0)
ZeroDivisionError: division by zero
```

NameError

```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in
<module>
    4 + spam*3
NameError: name 'spam' is not defined
```

TypeError

```
>>> '2' + 2
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    '2' + 2
TypeError: Can't convert 'int' object to str implicitly
```

ValueError

```
>>> int('one')
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int('one')
ValueError: invalid literal for int() with base 10: 'one'
```

Handling Exceptions

The simplest way to catch and handle exceptions is with a try-except block:

```
(x,y) = (5,0)
try:
    z = x/y
except ZeroDivisionError:
    print("divide by zero")
```

Try-Except (How it works I)

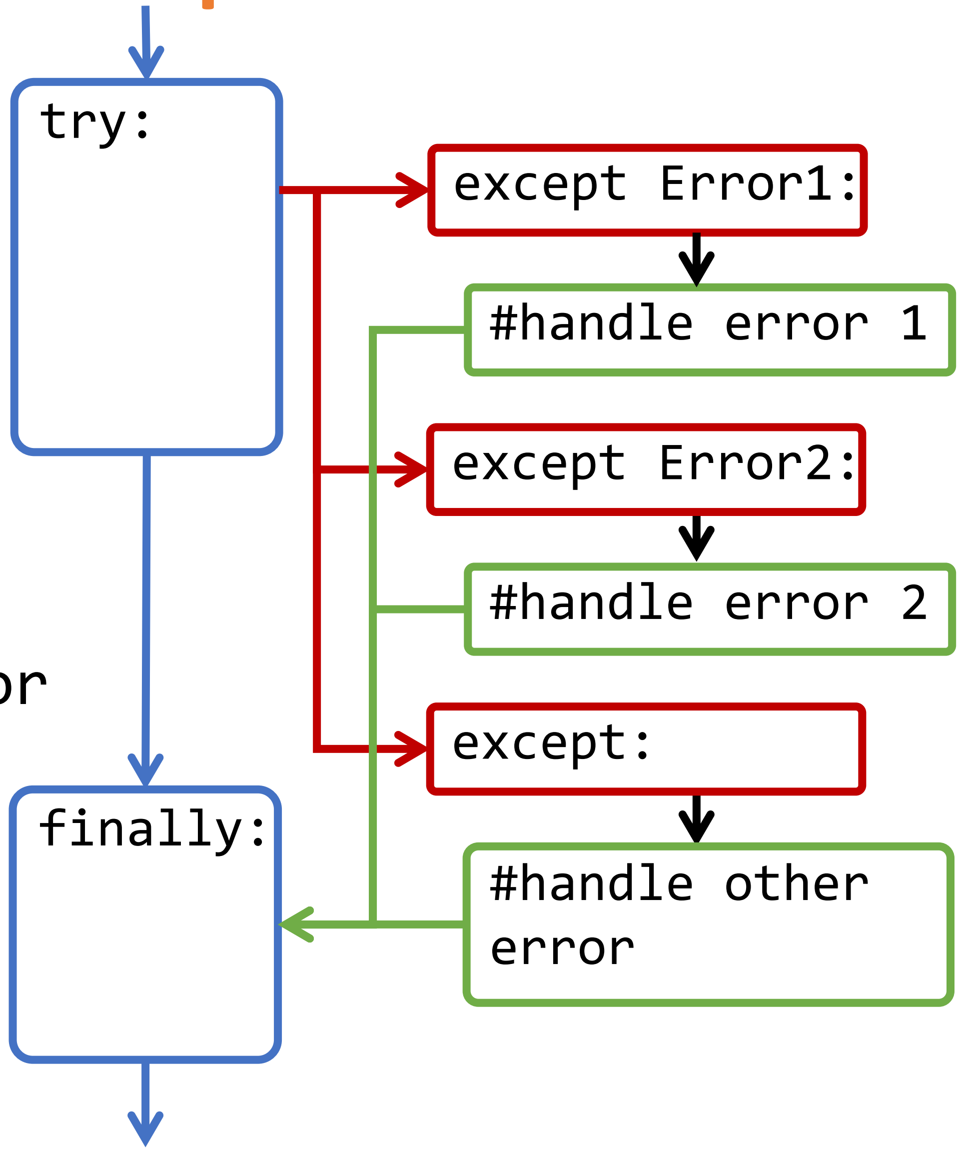
- The `try` clause is executed
- If an exception occurred, skip the rest of the `try` clause, to a matching `except` clause
- If no exception occurs, the `except` clause is skipped (go to the else clause, if it exists)
- The `finally` clause is always executed before leaving the `try` statement, whether an exception has occurred or not.

Try-Except

- A try clause may have more than 1 except clause, to specify handlers for different exception.
- At most one handler will be executed.
- Similar with if-elif-else
- finally will always be executed

Try-Except

```
try:
    # statements
except Error1:
    # handle error 1
except Error2:
    # handle error 2
except: # wildcard
    # handle other error
finally:
```



Try-Except Example

```
def divide_test(x, y):
    try:
      result = x / y
    except ZeroDivisionError:
      print "division by zero!"
    else:
      print "result is", result
    finally:
      print "executing finally clause"
```

Try-Except Blocks

```
>>> divide_test(2, 1)
result is 2.0
executing finally clause

>>> divide_test(2, 0)
division by zero!
executing finally clause

>>> divide_test("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

```
def divide_test(x, y):
    try:
      result = x / y
    except ZeroDivisionError:
      print "division by zero!"
    else:
      print "result is", result
    finally:
      print "executing finally
              clause"
```

Raising Exceptions

The raise statement allows the programmer to force a specific exception to occur:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

Exception Types

- Built-in Exceptions: http://docs.python.org/3.3/library/exceptions.html#bltin-exceptions
- User-defined Exceptions

User-defined Exceptions I

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
```

User-defined Exceptions II

```
try:
    raise MyError(2*2)
except MyError as e:
    print(‘Exception value:',
                e.value)
Exception value: 4

raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Why use Exceptions?

In the good old days of C, many procedures returned special ints for special conditions, i.e. -1

Why use Exceptions?

- But Exceptions are better because:
  - More natural
  - More easily extensible
  - Nested Exceptions for flexibility

Summary

- Memoization dramatically reduces computation.
  - Once a value is computed, it is remembered in a table (along with the argument that produced it).
  - The next time the procedure is called with the same argument, the value is simply retrieved from the table.
- memo_fib takes time = O(n)
- memo_choose takes ?? time?

Memoization vs Dynamic Programming

- Sometimes DP requires more computations
- DP requires the programmer to know exactly which entries need to be computed
- For smart programmer, DP can however be made more space efficient for some problems, i.e. limited history recurrences like Fibonacci